. A*
Justin Heyes-Jones
http://www.geocities.com/jheyesjones/astar.html

A* algorithm tutorial
Introduction
This document contains a description of the AI algorithm known as A*. The downloads section also has full source code for an easy to use extendable implementation of the algorithm, and two example problems.

Previously I felt that it would be wrong of me to provide source code, because I wanted to focus on teaching the reader how to implement the algorithm rather than just supplying a ready made package. I have now changed my mind, as I get many emails from people struggling to get something working. The example code is written in Standard C++ and uses STL, and does not do anything machine or operating system specific, so hopefully it will be quite useful to a wide audience.

State space search
A* is a type of search algorithm. Some problems can be solved by representing the world in the initial state, and then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

In this tutorial I will look at the use of state space search to find the shortest path between two points (pathfinding), and also to solve a simple sliding tile puzzle (the 8-puzzle). Let's look at some of the terms used in Artificial Intelligence when describing this state space search.

Some terminology

A node is a state that the problem's world can be in. In pathfinding a node would be just a 2d coordinate of where we are at the present time. In the 8-puzzle it is the positions of all the tiles.
Next all the nodes are arranged in a graph where links between nodes represent valid steps in solving the problem. These links are known as edges. In the 8-puzzle diagram the edges are shown as blue lines. See figure 1 below.
State space search, then, is solving a problem by beginning with the start state, and then for each node we expand all the nodes beneath it in the graph by applying all the possible moves that can be made at each point.

Heuristics and Algorithms

At this point we introduce an important concept, the heuristic. This is like an algorithm, but with a key difference. An algorithm is a set of steps which you can follow to solve a problem, which always works for valid input. For example you could probably write an algorithm yourself for multiplying two numbers together on paper. A heuristic is not guaranteed to work but is useful in that it may solve a problem for which there is no algorithm.
We need a heuristic to help us cut down on this huge search problem. What we need is to use our heuristic at each node to make an estimate of how far we are from the goal. In pathfinding we know exactly how far we are, because we know how far we can move each step, and we can calculate the exact distance to the goal.
But the 8-puzzle is more difficult. There is no known algorithm for calculating from a given position how many moves it will take to get to the goal state. So various heuristics have been devised. The best one that I know of is known as the Nilsson score which leads fairly directly to the goal most of the time, as we shall see.

Cost

When looking at each node in the graph, we now have an idea of a heuristic, which can estimate how close the state is to the goal. Another important consideration is the cost of getting to where we are. In the case of pathfinding we often assign a movement cost to each square. The cost is the same then the cost of each square is one. If we wanted to differentiate between terrain types we may give higher costs to grass and mud than to newly made road. When looking at a node we want to add up the cost of what it took to get here, and this is simply the sum of the cost of this node and all those that are above it in the graph.

8 Puzzle
Let's look at the 8 puzzle in more detail. This is a simple sliding tile puzzle on a 3*3 grid where one tile is missing and you can move the other tiles into the gap until you get the puzzle into the goal position. See figure 1.

Figure 1 : The 8-Puzzle state space for a very simple example

There are 362,880 different states that the puzzle can be in, and to find a solution the search has to find a route through them. From most positions of the search the number of edges (that's the blue lines) is two. That means that the number of nodes you have in each level of the search is 2^d where d is the depth. If the number of steps to solve a particular state is 18, then that’s 262,144 nodes just at that level.

The 8 puzzle game state is as simple as representing a list of the 9 squares and what's in them. Here are two states for example; the last one is the GOAL

state, at which point we've found the solution. The first is a jumbled up example that you may start from.

Start state SPACE, A, C, H, B, G, F, E

Goal state A, B, C, H, SPACE, D, G, F, E

The rules that you can apply to the puzzle are also simple. If there is a blank tile above, below, to the left or to the right of a given tile, then you can move that tile into the space. To solve the puzzle you need to find the path from the start state, through the graph down to the goal state.

There is example code to to solve the 8-puzzle on the downloads page.

Pathfinding

In a video game, or some other pathfinding scenario, you want to search a state space and find out how to get from somewhere you are to somewhere you want to be, without bumping into walls or going too far. For reasons we will see later, the A* algorithm will not only find a path, if there is one, but it will find the shortest path. A state in pathfinding is simply a position in the world. In the example of a maze game like Pacman you can represent where everything is using a simple 2d grid. The start state for a ghost say, would be the 2d coordinate of where the ghost is at the start of the search. The goal state would be where pacman is so we can go and eat him. There is also example code to do pathfinding on the downloads page.

Figure 2 : The first three steps of a pathfinding state space

Implementing A*

We are now ready to look at the operation of the A* algorithm. What we need to do is start with the goal state and then generate the graph downwards from there. Let's take the 8-puzzle in figure 1. We ask how many moves can we make from the start state? The answer is 2, there are two directions we can move the blank tile, and so our graph expands.

If we were just to continue blindly generating successors to each node, we could potentially fill the computer's memory before we found the goal node. Obviously we need to remember the best nodes and search those first. We also need to remember the nodes that we have expanded already, so that we don't expand the same state repeatedly.

Let's start with the OPEN list. This is where we will remember which nodes we haven't yet expanded. When the algorithm begins the start state is placed on the open list, it is the only state we know about and we have not expanded it. So we will expand the nodes from the start and put those on the OPEN list too. Now we are done with the start node and we will put that on the CLOSED list. The CLOSED list is a list of nodes that we have expanded.

f = g + h

Using the OPEN and CLOSED list lets us be more selective about what we look at next in the search. We want to look at the best nodes first. We will give each node a score on how good we think it is. This score should be thought of as the cost of getting from the node to the goal plus the cost of getting to where we are. Traditionally this has been represented by the letters f, g and h. 'g' is the sum of all the costs it took to get here, 'h' is our heuristic function, the estimate of what it will take to get to the goal. 'f' is the sum of these two. We will store each of these in our nodes.
Using the f, g and h values the A* algorithm will be directed, subject to conditions we will look at further on, towards the goal and will find it in the shortest route possible.

So far we have looked at the components of the A*, let's see how they all fit together to make the algorithm :

Pseudocode

[dennis]
Choose the node having the lowest f value.
If you arrive at the goal state, then you are done.
Repeat.

[dennis]
Theorem: Provided the h is never more expensive than the
actual cost to the goal state, A* will find a least cost path.
Proof: The best path will have the lowest cost and will be tried first.
Why? Recall that f = g + h. g is the cost to a node and h is less than or equal to the cost to the end state. Suppose the minimum cost to the end state is C along some path P. Then everywhere along the path P, the f value will be at most C. If some other path P' costs C' where C' > C, then before taking the last edge to the final solution node along P', we will take another node along P.

Hopefully the ideas we looked at in the preceding paragraphs will now click into place as we look at the A* algorithm pseudocode. You may find it helpful to print this out or leave the window open while we discuss it.

To help make the operation of the algorithm clear we will look again at the 8-puzzle problem in figure 1 above. Figure 3 below shows the f,g and h scores for each of the tiles.

Figure 3 : 8-Puzzle state space showing f,g,h scores

First of all look at the g score for each node. This is the cost of what it took to get from the start to that node. So in the picture the center number is g. As you can see it increases by one at each level. In some problems the cost may vary for different state changes. For example in pathfinding there is sometimes a type of terrain that costs more than other types.
Next look at the last number in each triple. This is h, the heuristic score. As I mentioned above I am using a heuristic known as Nilsson's Sequence, which converges quickly to a correct solution in many cases. Here is how you calculate this score for a given 8-puzzle state :

Nilsson's sequence score
A tile in the center scores 1 (since it should be empty)
For each tile not in the center, if the tile clockwise to it is not the one that should be clockwise to it then score 2
Finally add the total distance you need to move each tile back to its correct position

Looking at the picture you should satisfy yourself that the h scores are correct according to this algorithm.

Finally look at the digit on the left, the f score. This is the sum of g and h, and by tracking the lowest f down through the state space you are doing what the A* algorithm would be doing during its search.

fitness function: good one and bad one. Suppose the value we are looking one is one in which the conjunction of n boolean attributes is 1.
Obviously, all 1s is the answer, but if you were searching and your fitness function directly tested the conjunction, you would always get 0 unless you hit the right answer.
Better to have the fitness function point you in the right direction.

==========
p. 189: Travelling Salesman problem:
You are given a graph S where each edge has a cost.
The travelling salesman problem is to find a path through the graph that visits every node and returns to its origin and is the cheapest possible.

Christofides algorithm: compute a minimum spanning tree of S, MST(S).
Now, visit every node around the outside of the tree.
Interior nodes will be visited twice.
Still the total path length is less than twice the optimum.
Why? Because the minimum spanning tree must be shorter than any

```
travelling salesman path since that path defines a tree plus
an extra edge.

But we can do better:
Compute a minimum-length matching on the odd-degree vertices P of MST(S).
A minimum-length matching on P is a set of (new, not in S) edges E
where each edge in E touches a pair of points in P and
every point in P is touched by exactly one edge in E
and it is of minimum-length (a polynomial algorithm).
(Every tree has an even number of nodes that are odd-degree.
Single node has even degree.
Tree with two nodes has two odd-degree nodes.
Suppose I have a tree with this property and I add an edge from
n to m. If both n and m have odd degree, then they both get even degree.
If they both have even degree, then they both get odd degree and if
it's 1 and 1 then they remain 1 and 1. In every case, the invariant holds.)
The matching cannot be more than 1/2 of the optimum travelling
salesman path.

Here is why: consider the optimum travelling salesman path.
That path must include the nodes in this matching as well as perhaps others.
Shorten the path by ignoring nodes outside the matching nodes (by the
triangle inequality, this must reduce the cost).
Now consider alternate edges in the optimal tour among the
matching nodes.
Each set of alternate edges gives a matching.
Choose the one whose total distance is smaller.
That is less than half the distance of the shortened path
and the minimum matching can have no larger a distance.
Now union E to the MST(S) to get newEdge.
Every node has even degree in the graph defined by newEdge
and it has an Euler tour (a path that passes through every edge exactly
once).
Can shorten this even further by changing i,j and j,k to i,k
if j has already been visited.

The minimal length matching (also known as a minimum-weight
perfect matching) can
be computed in polynomial time using linear programming:
http://www2.isye.gatech.edu/~wcook/papers/IJOCmat.pdf


So, total length is at worst 3/2 optimal travelling salesman path.
```